Introduction to Static Code Analysis

What is Static Code Analysis?

- **Definition**: A process of analysing source code without executing it to find potential errors, vulnerabilities, and coding violations.
- **Purpose**: It helps identify weaknesses early in the development process, preventing security vulnerabilities, bugs, and performance issues before they reach production.
- **Key Metrics**:
  - Code complexity
  - Code duplication
  - Code smells (potential issues in the code)

▼ Code Complexity

- **Definition**:
  Code complexity refers to how difficult it is to understand, test, and maintain a codebase. It's often measured using

Cyclomatic Complexity, which counts the number of independent paths through a program's source code.

- **Measurement**:
  Cyclomatic complexity counts the number of decision points (like `if`, `while`, and `for` statements) in the code. The higher the complexity, the more testing and resources are required to ensure stability.
- **Why It Matters**:
  - **High complexity** indicates that a piece of code has many decision points, which can increase the likelihood of bugs, make it harder to test, and make the code more prone to errors.
  - **Low complexity** is often associated with cleaner, more maintainable code.
- **Best Practices**:
  - Refactor large methods into smaller, more manageable functions.
  - Keep functions and classes single-responsibility (one purpose per class/method).
- **Example**:

```
if (x > 0) {
    if (y < 0) {
        // Code block
    } else {
        // Another block
    }
}
```

  In this example, multiple paths (branches) through the code increase its complexity.

▼ Code Duplication

- **Definition**:
  Code duplication occurs when the same block of code appears multiple times in a codebase. This leads to redundancy, increased maintenance costs, and the potential for inconsistency.
- **Why It Matters**:

- Duplicated code is a **code smell** that increases the risk of errors during future modifications.
- If a bug is found in one piece of duplicated code, all other copies of that code need to be fixed as well.
- It leads to code that is harder to refactor, test, and maintain.

- **Best Practices**:
  - **DRY Principle (Don't Repeat Yourself)**: Refactor duplicated code into reusable functions or methods.
  - Use inheritance or composition to remove repeated logic between classes.
- **Example**:

```
int calculateSum(int a, int b) {
    return a + b;
}

int calculateProduct(int a, int b) {
    return a * b;
}
```

  These methods can be refactored to a more generic function to avoid code repetition.

▼ Code Smells

- **Definition**:
  Code smells are
  symptoms in the source code that might indicate deeper problems. They aren't necessarily bugs, but they make code harder to maintain and evolve over time. **Code smells** often point to violations of coding principles or patterns that might lead to future issues.
- **Common Examples**:
  1. **Long Methods**: Methods that do too much, making them hard to read, test, and maintain.
  2. **Large Classes**: Classes that handle too many responsibilities, making them complex.

3. **Primitive Obsession**: Overuse of basic data types instead of creating more meaningful classes.
4. **Feature Envy**: A method in one class that heavily relies on the internal data of another class.
5. **God Object**: A class that knows too much or does too much, violating the **Single Responsibility Principle**.

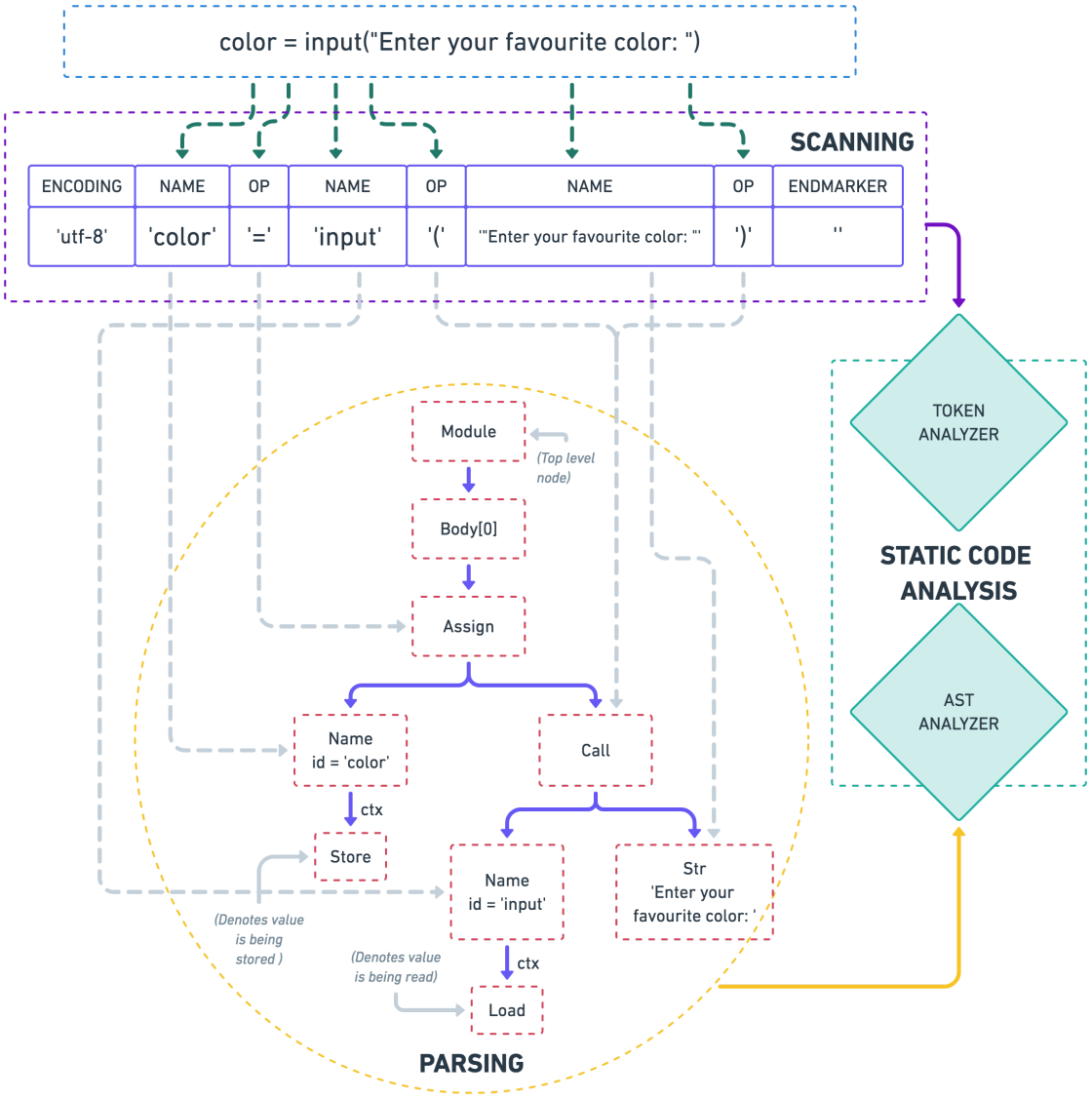
- **Why It Matters**:
  Code smells indicate areas that need refactoring. While they might not be immediate bugs, they make the code harder to manage in the long run, and they are often signs of
  technical debt.
- **Best Practices**:
  - Refactor large classes and methods into smaller, focused components.
  - Apply design patterns like **Strategy**, **Factory**, or **Observer** to fix specific code smells.
  - Follow SOLID principles (Single Responsibility, Open-Closed, etc.) to avoid common smells.
- **Example**:

```
class Order {
    public void addItem(Item item) {
        // Logic to add item to order
    }

    public void printInvoice() {
        // Logic to print the invoice
    }
}
```

  In this example, the `Order` class is handling both order management and invoice printing, violating **Single Responsibility**. This is a code smell.

▼ How Static Code Analysis Works



- **Scanning**: Automated tools analyze source code for predefined patterns and rules.
- **Analysis**: Tools break down the code and check for common vulnerabilities like SQL injection, XSS, buffer overflows, etc.
- **Reporting**: The analysis generates reports categorizing findings as bugs, code smells, vulnerabilities, and performance issues.

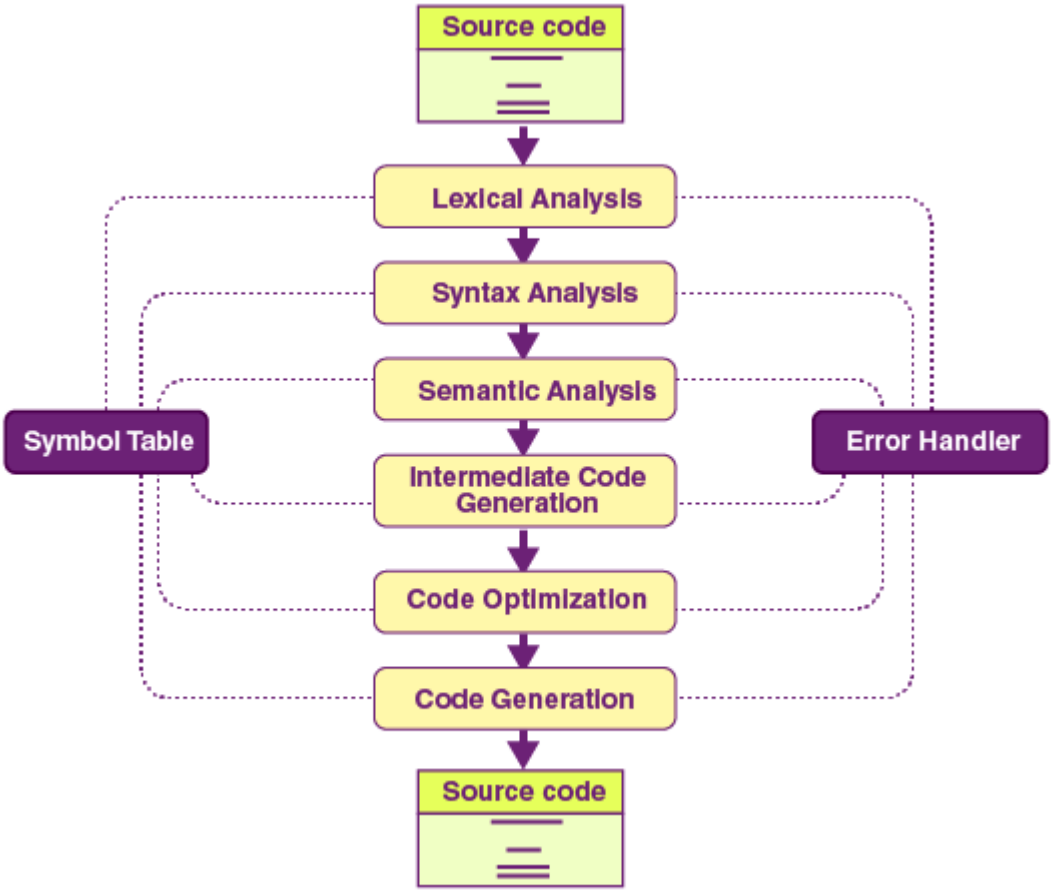
▼ Key Benefits of Static Code Analysis

1. **Early Bug Detection**: Reduces future costs by catching bugs early.
2. **Improved Code Quality**: Ensures adherence to coding standards and best practices.

3. **Security Improvements**: Identifies security vulnerabilities like SQL injection or XSS.
4. **Efficiency**: Automated and continuous scanning of large codebases.
5. **CI/CD Integration**: Seamless integration with DevOps pipelines.

▼ Types of Static Code Analysis

1. **Lexical Analysis**: Analyzes tokens and basic structures like variables and operators.
2. **Syntax Analysis**: Examines the structure of the code and syntax rules.
3. **Semantic Analysis**: Ensures that the logic of the code makes sense, e.g., variable assignments and returns.
4. **Data Flow Analysis**: Examines how data flows through the program to identify unused variables or null pointer exceptions.



▼ Tools for Static Code Analysis

1. `SonarQube :`

   - **Language Support**: Java, JavaScript, Python, etc.
   - **Key Features**: Bugs, vulnerabilities, code smells, technical debt.

2. `ESLint`:

   - **Language**: JavaScript.
   - **Key Features**: Style guide enforcement, best practices.

3. `Checkmarx`:

   - **Focus**: Security.
   - **Key Features**: Vulnerabilities, OWASP Top 10 compliance.

4. `FindBugs` / `SpotBugs`:

- **Language**: Java.
- **Key Features**: Bug patterns in Java bytecode.

▼ Best Practices for Static Code Analysis

1. **Integrate Early**: Apply static analysis from the beginning of the development cycle.

   > 💡 Cisco have reduced their defect rates significantly by integrating static code analysis early in their development cycle (by up to 50%). This prevents security vulnerabilities and other issues from progressing to later stages of development, where they become more costly to fix.

2. **Automate in CI/CD Pipelines**: Ensure continuous scanning for every commit or build.

   > 💡 IBM's research shows that detecting and fixing defects during development is up to 100x cheaper than post-release.

3. **Review Findings Regularly**: Don’t ignore reports; address critical issues promptly.

> 💡 Around 65% of developers delayed addressing static code analysis findings for days. Prioritising critical issues and reviewing findings regularly can improve security and stability by addressing vulnerabilities like SQL injection or XSS early. Google, for instance, mandates regular code reviews with high priority given to security bugs.

4. **Tune Rules**: Customise analysis rules based on the project requirements.

> 💡 A case study from Adobe illustrates that fine-tuning analysis rules helps reduce the noise in reports, making them more actionable. They adjusted their tool's configuration to better align with their coding standards, significantly reducing unnecessary alerts.

5. **Complement with Manual Code Reviews**: Tools can't detect everything; manual reviews are still necessary.

> 💡 Facebook and Amazon have formalized peer code reviews, ensuring that human reviewers catch potential issues that tools might miss. Code improved by peer examination typically suffers 80% fewer defects over its lifetime.

▼ Limitations of Static Code Analysis

1. **False Positives**: Not all flagged issues are real problems, requiring manual inspection.

💡

> A report from NIST suggests that static analysis tools often generate false positives, which can account for 10-30% of the issues flagged. E.g., Facebook experienced high false-positive rates before tuning their static analysis rules.

> A team at Microsoft found that fine-tuning tools like SonarQube for their specific coding environment significantly reduced false positives, which in turn boosted developer trust in the tool.

2. **No Runtime Error Detection**: Unlike dynamic analysis, it can't identify errors that only occur during execution.

💡

> Dynamic analysis tools like Valgrind and runtime monitoring are often used to complement static analysis to catch these kinds of problems in production systems.

3. **Context Ignorance**: Might not understand the broader context of the code.

> A study by the University of Maryland showed that static analysis tools often struggle to understand the full context of a codebase, especially in complex architectures where global variables and interdependent modules are prevalent. This is one of the reasons why some companies, like Google, supplement static analysis with manual reviews to understand the context more clearly.

> At Uber, engineers found that static analysis tools couldn't fully understand the context of micro-service interactions, leading them to implement additional dynamic analysis in the testing phase.

4. **Performance Overhead**: Large codebases can take time to analyse thoroughly.

> Salesforce reported that the time required to run static analysis was significant, particularly in continuous integration workflows. To combat this, they parallelised their scans to reduce performance bottlenecks.

> Facebook engineers have reported that analysing large-scale code can sometimes slow down CI pipelines, leading to performance issues. They had to develop their own tools (Infer, for example) to handle the load efficiently.